How Efficient can Memory Checking be?

Cynthia Dwork* Moni Naor† Guy N. Rothblum‡ Vinod Vaikuntanathan§

Abstract

We consider the problem of memory checking, where a user wants to maintain a large database on a remote server but has only limited local storage. The user wants to use the small (but trusted and secret) local storage to detect faults in the large (but public and untrusted) remote storage. A memory checker receives from the user store and retrieve operations to the large database. The checker makes its own requests to the (untrusted) remote storage and receives answers to these requests. It then uses these responses, together with its small private and reliable local memory, to ascertain that all requests were answered correctly, or to report faults in the remote storage (the public memory).

A fruitful line of research investigates the complexity of memory checking in terms of the number of queries the checker issues per user request (query complexity) and the size of the reliable local memory (space complexity). Blum et al., who first formalized the question, distinguished between online checkers (that report faults as soon as they occur) and offline checkers (that report faults only at the end of a long sequence of operations). In this work we revisit the question of memory checking, asking *how efficient can memory checking be?*

For online checkers, Blum et al. provided a checker with logarithmic query complexity in n, the database size. Our main result is a lower bound: we show that for checkers that access the remote storage in a deterministic and non-adaptive manner (as do all known memory checkers), their query complexity must be at least $\Omega(\log n / \log\log n)$. To cope with this negative result, we show how to trade off the read and write complexity of online memory checkers: for any desired logarithm base d, we construct an online checker where either reading *or* writing is inexpensive and has query complexity $O(\log_d n)$. The price for this is that the other operation (write or read respectively) has query complexity $O(d \cdot \log_d n)$. Finally, if even this performance is unacceptable, *offline* memory checking may be an inexpensive alternative. We provide a scheme with $O(1)$ amortized query complexity, improving Blum et al.'s construction, which only had such performance for long sequences of at least n operations.

*Microsoft Research, Silicon Valley Campus, 1065 La Avenida Mountain View, CA. 94043 USA; email: dwork@microsoft.com

†Incumbent of the Judith Kleeman Professorial Chair, Department of Computer Science and Applied Math, The Weizmann Institute of Science, Rehovot 76100, Israel; email: moni.naor@weizmann.ac.il. Research supported in part by a grant from the Israel Science Foundation.

‡MIT, Cambridge, USA; email: rothblum@csail.mit.edu. Research supported by NSF Grants CCF-0635297, NSF-0729011, CNS-0430336, Israel Science Foundation Grant 700/08 and by a Symantec Graduate Fellowship.

§IBM Research. Supported in part by NSF grants CCF-0635297 and Israel Science Foundation 700/08.

1 Introduction

Consider a user who wants to maintain a large database but has only limited local storage. A natural approach is for the user to store the database on a remote storage server. This solution, however, requires that the user trust the remote storage server to store the information reliably. It is natural to ask whether the user can use his or her small (but trusted and secret) local storage to detect faults in the large (but public and untrusted) remote storage. This is the problem of *memory checking*, as introduced by Blum, Evans, Gemmel, Kannan and Naor [BEG+94] in 1991. Since then, this problem has gained even more importance for real-world applications, see for example the more recent works of Clarke *et al.* [CSG+05], Ateniese *et al.* [ABC+07], Juels and Kaliski [JK07], Oprea and Reiter [OR07] and Shacham and Waters [SW08]. Large databases are increasingly being outsourced to untrusted storage providers, and this is happening even with medical or other databases where reliability is crucial. Another wide-spread and growing phenomenon are services that offer individual users huge and growing remote storage capacities (e.g. webmail providers, social networks, repositories of digital photographs, etc.). In all of these applications it is important to guarantee the integrity of the remotely stored data.

Blum *et al.* formalized the above problem as the problem of memory checking. A memory checker can be thought of as a layer between the user and the remote storage. The checker receives from its user a sequence of "store" and "retrieve" operations to a large unreliable memory. Based on these "store" and "retrieve" requests, it makes its own requests to the (untrusted) remote storage and receives answers to these requests. The checker then uses these responses, together with a small private and reliable "local" memory, to ascertain that all requests were answered correctly, or to report that the remote storage (the *public memory*) was faulty. The checker's assertion should be correct with high probability (a small two-sided error is permitted). Blum *et al.* made the distinction between online and offline memory checking. An *online* checker verifies the correctness of each answer it gives to the user. An *offline* checker gives only the relaxed guarantee that after a (long) sequence of operations a user can verify whether or not there was an error *somewhere* in the sequence of operations. Two important complexity measures of a memory checker are its *space complexity*, the size of the secret reliable "local" memory, and its *query complexity*, the number of queries made to the unreliable memory per user request. One may consider additional complexity measures such as the alphabet size (the size of words in the public memory), and more measures such the checker's and public memory's running times, the amount of public storage, etc. See Section 2 for formal definitions and a fuller discussion.

In this work we revisit the question of designing efficient memory checkers. Our main result is a lower bound on the query complexity of deterministic and non-adaptive online memory checkers. We also present new upper bounds for both online and off-line memory checking.

Online Memory Checkers. The strong verification guarantee given by online memory checkers makes them particularly appealing for a wide variety of applications. Blum *et al.* construct efficient online memory checkers with space complexity that is proportional to the size of a cryptographic key, and *logarithmic* query complexity. Their construction(s) assume that a one-way function exists and that the adversary who controls the public memory is efficient and cannot invert the function. In fact, this assumption was shown to be *essential* by Naor and Rothblum [NR05], who showed that any online memory checker with a non-trivial query-space tradeoff can only be computationally secure and must be based on the existence of a one-way function. Even in the computational setting, the space complexity of Blum *et al.*'s online memory checkers is intuitively optimal, since if the secret memory is s bits long, an (efficient) adversary can guess it (and fool the memory checker) with probability at least 2^{-s}. What is less clear, however, is whether the logarithmic query complexity is essential (in a computational setting). This is an important question, since while this logarithmic overhead is reasonable, in

many applications it remains a significant price to have to pay for data verification.

Where then does this overhead come from? The logarithmic query complexity is needed to avoid *replay attacks*, in which the correct public memory is swapped for some older version of it. In most applications replay attacks are a serious threat, and Blum *et al.* (and all other solutions we know of) use a tree structure to overcome this devastating class of attacks. This tree structure incurs a logarithmic overhead which is basically the depth of the tree. We begin by asking whether it is possible to avoid the logarithmic overhead and construct memory checkers with lower query complexity. We show that the answer is negative (even in the cryptographic setting!) for all known and/or practical methods of designing memory checkers.

A Query Complexity Lower Bound. Consider online memory checkers, where for each store or retrieve request made by the user, the locations that the checker accesses in the public memory are fixed and known. We call such a checker a deterministic and non-adaptive checker. Known checker constructions are all deterministic and non-adaptive, indeed tree authentication structures all have this property. Our main result is a new lower bound, showing that any deterministic non-adaptive memory checker must have query complexity $\Omega(\log n / \log\log n)$. Thus the logarithmic query complexity overhead is (almost) unavoidable for online memory checking. This is stated more fully (but still informally) below, see Section 3 for the full details.

Theorem 3.1 Let $\mathcal{C}$ be a non-adaptive and deterministic memory checker for an n-index boolean database, with space complexity $s \leq n^{1-\varepsilon}$ for some $\varepsilon > 0$, query complexity q and a polylog-length alphabet (public memory word size). It must be that $q = \Omega(\frac{\log n}{\log\log n})$.

Let us examine the above theorem more closely. Considering only checkers that are deterministic and non-adaptive may seem at first glance to be quite restrictive. We argue, however, that *practical* checkers will likely have to conform to this restriction:

An *adaptive* checker is one that chooses sequentially which locations in the remote storage it reads and writes, and chooses these locations based on the contents of earlier read locations. This means that the checker needs to conduct, for every user request, several rounds of communication with the remote storage (the checker needs to know the contents of a location before deciding which location it accesses next). Since this communication happens over a network, it may very well lead to latency which results in more of an overhead than the logarithmic query complexity of non-adaptive checkers. In addition, in cases where the memory contents are encrypted non-adaptive memory access may be especially desirable, as the set of locations accessed reveals nothing about the (decrypted) contents of the memory.

Another problem with adaptive checkers is that they make *caching* the results much more difficult, since the actual locations needed to be stored in the faster memory change between accesses.

A *non-deterministic* checker may also result in worse performance. Such a checker strategy, with queries that are either significantly randomized or hard to predict (depending on the secret memory), destroys locality in the user's queries and makes it hard to utilize caching mechanisms. In particular, user accesses to neighboring database indices would not necessarily be mapped to checker accesses to neighboring locations in the remote storage, and repeated user accesses to the same database index would not necessarily be mapped to the same locations in the remote storage. For many of the applications of memory checking, this will result in an unacceptable overhead for the remote storage server. We note that Blum *et al.*'s constructions, as well as all of the constructions we present in this work, have the important property that they do preserve (to a large extent) the locality of a user's data accesses.

Finally, we note that the restriction on sub-linear space is essential, as the problem of memory checking makes very little sense with linear secret memory; the checker can simply store the entire database in reliable memory! Finally, it is interesting to ask whether the lower bound can

be extended to larger alphabets (we focus on polylog word lengths or quasi-polynomial alphabet size). We do note that the best parameters attained both in our work and in [BEG+94] can be attained with words of poly-logarithmic length.

Trading Off Reads and Writes. Is all hope of improving the performance of online memory checkers lost in light of Theorem 3.1? We argue that this is not the case. While we cannot improve the query complexity of online checkers beyond logarithmic, we observe that in many applications read operations are far more numerous than write, and vice versa. One example for frequent read operation is a database that is read frequently but updated only periodically. An example for frequent write is a repository of observed data (say climate measurements) that is constantly updated but polled much less frequently.

For these settings we show how to trade off the query complexity of read and write operations. For any desired logarithm base d, we show how to build an online checker where the frequent operation (read or write) is inexpensive and has query complexity $O(\log_d n)$, and the infrequent operation (write or read respectively) has query complexity $O(d \cdot \log_d n)$. The space complexity is proportional to a security parameter (it can be poly-logarithmic under an exponential hardness assumption), and the alphabet size is the logarithm of the desired soundness. The construction uses a pseudo-random function (see [GGM86]), and can thus be based on the existence of any one-way function. This means, for example, that if one is willing to have a polynomial (n^{ε}) write complexity, then we can get a constant ($O(1/\varepsilon)$) read complexity (and vice versa). This may be very useful for a database that is read frequently but only updated infrequently.

To achieve this tradeoff, we provide two constructions: one for efficient write and one for efficient read. Both of these use a tree-based authentication structure, where the tree's depth is $\log_d n$. The efficient-write construction can viewed as a generalization of Blum *et al.*'s tree-based online memory checker. The efficient-read construction is different in the way it stores authentication information. Intriguingly, we do not know how to get a good read-write trade-off based on UOWHFs where the checker's memory only needs to be reliable (and not necessarily private). Blum *et al.* were able to present such a construction (albeit with a nearly exponential-size alphabet) with logarithmic query complexity, but their construction does not easily yield itself to a read-write tradeoff. See Section 4 for the full details.

While we believe that these trade-offs are very useful for many applications, we still cannot beat the lower bound of Theorem 3.1: the *sum* of read and write complexities is still at least logarithmic in n (not surprisingly, since the above checkers are deterministic and non-adaptive). For many other applications this may still be prohibitively expensive. This leads us then to revisit Blum *et al.*'s notion of *offline memory checking*, where the verification guarantee of the checker is weaker, but it is possible to achieve better performance.

An Off-Line Alternative. Blum *et al.* suggested the notion of an *offline* memory checker. Such a memory checker gives the relaxed guarantee that after a (long) sequence of operations it can be used to check whether there was an error. In other words, whether any value retrieved from public memory was different from the last value stored at that location. The advantage of offline memory checkers is that they allow much better parameters. Specifically, Blum *et al.* gave a construction where for any *long* sequence of user operations (at least n operations) the *amortized* query complexity is $O(1)$ and the space complexity is logarithmic in n and in the soundness parameter. Remarkably, the security of their checker is *information theoretic*, and does not rely on any cryptographic assumptions.

We conclude that for applications in which the offline guarantee suffices, say when the user does not mind that some of the data may be retrieved incorrectly as long as this is eventually detected, the query complexity of both read and write can be reduced to $O(1)$. It is natural to ask what can possibly be improved in the above construction, as the (amortized) query and space complexity seem optimal. One place for improvement is that Blum *et al.*'s construction is

highly *invasive*: the checker stores a significant amount of additional information in the public memory on top of the database. Ajtai [Ajt02] showed that this invasiveness cannot be avoided (see Appendix A for an overview of Ajtai's results).

We focus on a different parameter. The above off-line checker only guarantees good amortized performance for *long sequences* of at least n operations. We observe that for shorter operation sequences, the amortized performance will be quite bad, as their checker needs to always scan the *entire public memory* before deciding whether there were any errors. So for a k operation sequence, the amortized query complexity will be $O(n/k)$. In Section 5 we overcome this obstacle. We present a simple and inexpensive offline memory checker where the amortized query complexity for *any sequence of operations* (even a short one) is $O(1)$. Moreover, we show that similar ideas can be used to decrease the invasiveness of the checker, and that the invasiveness (the amount of extra information stored in public memory on top of the database) only needs to be proportional to the number of database locations that the checker actually accesses (instead of always being proportional to the entire database size as in Blum *at al.*). We note that we can overcome Ajtai's invasiveness lower bound in this setting because the proof of that lower bound considers sequences of operations that access every location in the database (again, see Appendix A for the details).

Organization. We begin in **Section 2** with definitions of memory checkers (we refer the reader to Goldreich [Gol01, Gol04] for standard cryptographic definitions). In **Section 3** we state and prove our lower bound for the query complexity of online memory checkers. Constructions of read-write tradeoffs are presented in **Section 4**. Finally, in **Section 5** we present a new and improved construction of offline checkers.

2 Memory Checkers: Definitions

A memory checker is a probabilistic Turing machine $\mathcal{C}$ with five tapes: a read-only input tape for receiving read/write requests from the user $\mathcal{U}$ to the RAM or database, a write-only output tape for sending responses back to the user, a read-write work tape (the secret reliable memory), a write-only tape for sending read/write requests to the memory $\mathcal{M}$ and a read only input tape for receiving $\mathcal{M}$'s responses.

Let n be the size of the database (the RAM) $\mathcal{U}$ is interested in using. A checker is presented with "store" (write) and "retrieve" (read) requests made by $\mathcal{U}$ to $\mathcal{M}$. After each "retrieve" request $\mathcal{C}$ returns an answer or outputs that $\mathcal{M}$'s operation is BUGGY. $\mathcal{C}$'s operation should be both correct and complete for all polynomial (in n) length request sequences. Formally, we say that a checker has completeness c (2/3 by default) and soundness s (1/3 by default) if:

- Completeness. For any polynomial-length sequence of $\mathcal{U}$-requests, as long as $\mathcal{M}$ answers all of $\mathcal{C}$'s "retrieve" requests correctly (with the last value that $\mathcal{C}$ stored at that location), $\mathcal{C}$ also answers all of $\mathcal{U}$'s "retrieve" requests correctly with probability at least c.[1]
- Soundness. For any polynomial-length sequence of $\mathcal{U}$-requests, for *any* (even incorrect or malicious) answers returned by $\mathcal{M}$, the probability that C answers a user request incorrectly is at most s. $\mathcal{C}$ may either recover the correct answer independently or answer that $\mathcal{M}$ is "BUGGY", but it may not answer a request incorrectly (beyond probability s).

Note that the completeness and soundness requirements are for *any* request sequence and for *any* behavior of the unreliable memory. Thus we think of $\mathcal{U}$ and $\mathcal{M}$ as being controlled by a malicious adversary. A memory checker is secure in the computational setting if the soundness property holds versus any PPTM adversary. In this setting, if one-way functions exist, then they can be used to construct very good online memory checkers (see [BEG+94]).

As previously noted, [BEG+94] make the distinction between memory checkers that are online and offline. An *offline* checker is notified before it receives the last "retrieve" request in a sequence of requests. It is only required that if at some point in the sequence a user retrieve request was answered incorrectly, then the checker outputs BUGGY (except with probability s). The task of an *online* checker is more difficult: if $\mathcal{M}$'s response to some request was incorrect, $\mathcal{C}$ must immediately detect the error or recover from it (with high probability). $\mathcal{C}$ is not allowed (beyond a small probability) to ever return an erroneous answer to $\mathcal{U}$. Note that after the memory checker informs the user that $\mathcal{M}$'s operation was BUGGY, there are no guarantees about the checker's answers to future queries.

Recall that the two important measures of the complexity of a memory checker are the size of its secret memory (*space complexity*) and the number of requests it makes per request made by the user (*query complexity*). The query complexity bounds the number of locations in public memory accessed (read or written) per user request. We would prefer memory checkers to have small space complexity and small query complexity. A memory checker is *polynomial time* if $\mathcal{C}$ is a PPTM (in n).

A **deterministic and non-adaptive** memory checker is a checker $\mathcal{C}$ where the locations it queries in public memory are set and depend (deterministically) only on the database index being stored or retrieved. We call such a checker non-adaptive because it chooses the entire list of locations to access in public memory without knowing the value of the public (or secret) memory at any location. We note, though, that even a non-adaptive checker can decide which *values* to write into those (non-adaptively chosen) locations in an adaptive manner, based on values it reads and the secret memory. One way to think of a deterministic non-adaptive checker is by associating with each index in the database a static set of locations that the checker accesses when storing or retrieving that index.

[1] In fact in all our constructions we get *perfect completeness*; the checker answers all requests correctly with probability 1.

Similarly, for a deterministic and non-adaptive checker, each location in the public memory can be associated with the set of database indices that "access" it. We say that a location in public memory is ***t*-heavy** if there are at least t database indices that access it (for store or retrieve requests).

We say that $\mathcal{C}$ is a $(\mathbf{\Sigma}, \mathbf{n}, \mathbf{q}, \mathbf{s})$**-checker** if it can be used to store a (binary) database of n indices with query complexity q and space complexity s, where the secret and public memory are over the alphabet Σ (we allow this alphabet to be non-binary).

3 Lower Bounds

Throughout this section we obtain a lower bound for memory checking by using **restrictions of memory checkers**. When we talk about restricting a memory checker to a subset of database indices, we start with a checker $\mathcal{C}$ say for databases with n indices, and obtain from it a checker $\mathcal{C}'$ for databases with $n' < n$ indices. This is done simply by selecting a subset I of the indices that $\mathcal{C}$ works on ($|I| = n'$) and ignoring all of the rest. Naturally, the completeness and soundness of $\mathcal{C}$ carry over to $\mathcal{C}'$. Intuitively, this may also mean that we can ignore some of the locations in public memory or some of the secret memory, but we make no such assumptions in this work. It may seem that this is a bad bargain: the number of indices is decreased without gaining anything. However, when performing the restrictions below we gain (reduce) something in other complexity measures such as the query complexity. Sometimes this will require making additional changes to the checker, such as moving some locations from public to secret memory.

We will assume without loss of generality that the read and the write operations access the same locations. This involves at most doubling the number of accesses per operation.

We now present our lower bound for non-adaptive and deterministic checkers.

Theorem 3.1. *Let $\mathcal{C}$ be a (Σ, n, q, s) deterministic and non-adaptive online memory checker, with $s \leq n^{1-\varepsilon}$ for some $\varepsilon > 0$ and $|\Sigma| \leq n^{\text{poly}\log n}$. It must be that $q = \Omega(\frac{\log n}{\log\log n})$.*

of Theorem 3.1. Let $q_0 = q$ be the query complexity of the checker $\mathcal{C}$. The proof proceeds by iteratively restricting the checker, gradually lowering its query complexity until a lower bound can be obtained. This is done by examining the memory checker and determining whether there is a relatively large set of "heavily queried" locations in the public memory. I.e. whether there is a polynomial size set of locations in the public memory, each of which is queried when reading or writing many database indices. Recall that we call such heavily-queried locations in the public memory "heavy locations".[2] If there is such a set of heavy locations, then those public memory locations are moved into the secret memory and the query complexity of the checker is reduced significantly. In this case we advance towards our goal of lower bounding the query complexity. This intuition is formalized by Lemma 3.1 (the proof appears below):

Lemma 3.1. *Let $\mathcal{C}$ be a (Σ, n, q, s) deterministic and non-adaptive online memory checker. For every threshold $t \in \mathbb{N}$ such that $n > t$ the following holds: If there exists $m \in \mathbb{N}$ such that there are m or more t/m-heavy locations in public memory, then for some $i \in [q]$ the memory checker $\mathcal{C}$ can be restricted to a $(\Sigma, t/2^{i+2}, q-i, s+m)$-checker.*

Lemma 3.1 is used iteratively as long as there are heavy public memory locations, restricting the memory checker to only a (large) subset of its indices while lowering its query complexity (q). This comes at the cost of only a modest drop in the number of indices (n) and a moderate increase in the space complexity (s). We repeat this iteratively, reducing the query complexity until there is no set of "heavy" locations in the public memory. If we can apply the lemma many times, then we get a lower bound on the checker's query complexity: each application of the lemma reduces the query complexity, so if we applied the lemma many times the initial query complexity had to have been high.

The reason that we can apply Lemma 3.1 many times is that otherwise we are left with a checker on many indices with no set of "heavy" locations. If there is no set of "heavy" public memory locations, then the (possibly reduced) public memory can be partitioned into relatively many parts that are *disjoint* in the sense that each part is queried only by a single index of the database. We can restrict the checker again, but this time to obtain a checker with many indices, relatively small secret memory and query complexity 1. This is formalized in Lemma 3.2 (proof below):

[2] I.e. locations accessed by many indices - more formally a location is *t-heavy* if there are t different queries $i \in [n]$ that access it.

Lemma 3.2. *Let $\mathcal{C}$ be a (Σ, n, q, s) deterministic and non-adaptive online memory checker. Then, for every $\alpha \in \mathbb{N}$ such that $\alpha < n$, and for every threshold $t \in \mathbb{N}$ such that $n > 4t \cdot q \cdot \log n$, the following holds:*

If for every *integer $m \in \{1, \ldots, \alpha\}$, there are fewer than m locations in public memory that are t/m-heavy, then the memory checker $\mathcal{C}$ can be restricted to a $(\Sigma^q, n \cdot \alpha/(2q \cdot t), 1, s/q)$-checker.*

Finally, we show that such a "disjoint" checker implies a contradiction. In particular, it must have space complexity that is more or less proportional to the number of disjoint parts. Unless the memory checker has already been restricted to very few indices (in which case we have a query complexity lower bound), this results in a contradiction, since the checker's space complexity is bounded (by a small polynomial in n). The intuition that a disjoint checker must have large space complexity is formalized in Lemma 3.3 (proof below):

Lemma 3.3. *Let $\mathcal{C}$ be a $(\Sigma, n, q = 1, s)$ deterministic and non-adaptive online memory checker, i.e. a checker that makes only a single query, where the location that each index queries in public memory is different. Then, $s \geq \frac{n}{\log |\Sigma|} - 1$.*

We postpone proving the lemmas and proceed with a formal analysis. We take $\alpha = n^d$, for a constant $0 < d < 1$ to be specified later. We iteratively examine and restrict the memory checker. Let $\mathcal{C}_i$ be the checker obtained after the i-th iteration ($\mathcal{C}_0 = \mathcal{C}$ is the original checker), let n_i be the number of indices in its database and s_i its space complexity. Taking a threshold $t_i = \frac{n_i}{\log^c n}$, where $c > 1$ is a constant specified below, we check whether or not the "new" checker $\mathcal{C}_i$ has a set of heavy indices in its public memory. We only iterate as long as $n_i > \alpha$. Formally, there are two possible cases:

1. If $\mathcal{C}_i$ has a set of $m \leq \alpha$ public memory locations that are at least t_i/m-heavy, then by Lemma 3.1:

   For some $j \in \{1, \ldots, q\}$, we can build from C_i a $(\Sigma, t_i/2^{j+2}, q - j, s_i + \alpha)$ deterministic and non-adaptive online memory checker $\mathcal{C}_{i+1}$.
2. If for every integer $m \leq \alpha$ the checker $\mathcal{C}_i$ does not have a set of m public memory locations that are t_i/m-heavy, and choosing c, d such that $n_i > 4t_i \cdot q \cdot \log \alpha$, by Lemma 3.2:

   We can build from C_i a $(\Sigma^q, n_i \cdot \alpha/(2q \cdot t_i), 1, s_i/q)$ deterministic and non-adaptive online memory checker. If n_i is reasonably large, i.e. has not been reduced by repeated iterations of Case 1, then this will imply a contradiction.

Recall that q_0 denotes the query complexity of the initial checker $\mathcal{C}$, before any application of Lemmas 3.1 and 3.2. Assume for a contradiction that $q_0 \leq \log n/(3c \cdot \log \log n)$. Let $j \in [q + 1]$ be the *total* number of queries reduced by the iterative applications of Lemma 3.1, *i.e.*, the number of queries reduced by the iterations in which Case 1 occurred. Since we assumed $q \leq \log n/(3c \cdot \log \log n)$, we know that $j < \log n/(3c \cdot \log \log n)$. Thus, in the first iteration in which Case 2 applies (say the i-th iteration in total), it must be the case that

$$n_i \geq n/(\log^{c \cdot j} n \cdot 2^{3 \log n/3c \cdot \log \log n}) = n/(\log^{c \cdot j} n \cdot 2^{\log n/c \log \log n}) > n^{1-\varepsilon/2}.$$

Recall that we only iterate so long as $n_i > \alpha$, so we can choose any $\alpha < n^{1-\varepsilon/2}$. The space s_i used by this restricted checker is at most $s + i \cdot \alpha \leq s + \log n \cdot \alpha$. As usual, $t_i = n_i/\log^c n$, and choosing $c > 2$ we get that

$$4t_i \cdot q \cdot \log \alpha \leq n_i/(\log^c n \cdot \log n \cdot d \log n) < n_i$$

Applying Lemma 3.2, we obtain a $(\Sigma^q, n_i \cdot \alpha/(2q \cdot t_i), 1, s_i/q)$-checker. Now, by Lemma 3.3, which bounds the space complexity of one-query checkers, we get that it must be the case that:

$$s_i \geq n_i \cdot \alpha/(2q \cdot t_i \cdot \log |\Sigma|) \geq \log^{c-1} n \cdot \alpha/(2 \log |\Sigma|)$$

But on the other hand we know that

$$s_i \leq s + \log n \cdot \alpha.$$

We know $|\Sigma| \leq 2^{\text{poly}\log n}$, and choose c such that $\log^{c-1} n/(2\log|\Sigma|) > 2\log n$. We also set $\alpha > 2s = 2n^{1-\varepsilon}$. Recall that we also needed $\alpha < n_i$, but this is fine since $n_i > n^{1-\varepsilon/2}$. In conclusion, we set α by choosing d such that $1-\varepsilon < d < 1-\varepsilon/2$, i.e. such that

$$2s = 2n^{1-\varepsilon} < \alpha = n^d < n_i = n^{1-\varepsilon/2}$$

We get that

$$s > \log^{c-1} n \cdot \alpha/(2 \cdot \log|\Sigma|) - \log n \cdot \alpha > \log n \cdot \alpha > 2s$$

This is a contradiction! □

of Lemma 3.1. If there is a set M of m locations in public memory that are all t/m-heavy (i.e. each accessed by at least t/m indices), then we "restrict" the memory checker to only work for some of the indices that access one or more of the heavy locations. Let $I \subseteq [n]$ be the set of database indices that access at least one of the locations in M (the "heavy" locations).

We claim that for some $i \in \{1, \ldots, q\}$, there are at least $t/2^{i+2}$ indices in I that each access at least i locations in M. To see this, assume for a contradiction that this is not the case. Then the sum of the number of locations in M that are accessed by each database index (and in particular by the indices in I) is less than:

$$\sum_{i=1}^{q} i \cdot t/2^{i+2} = t \cdot \sum_{i=1}^{q} i/2^{i+2} < t$$

On the other hand, since there are m locations in M that are at least t/m-heavy, the sum of locations in M read by database indices must be at least t and we get a contradiction.

We restrict the checker to the indices in I that read at least i locations in M, and move these locations to the secret memory. This increases the space complexity (size of the secret memory) from s to $s+m$. By the above, there are at least $t/2^{i+2}$ such indices. For each of them, we have reduced their query complexity from q to $q-i$. The alphabet size remains unchanged. □

of Lemma 3.2. If there are only a few relatively heavy locations in the public memory, then we eliminate indices and split the public memory in "disjoint chunks": subsets of the public memory that are disjoint in the sense that no location in any chunk is accessed by two different indices. This is done in a greedy iterative manner. We go over the locations in public memory one by one; for each of them we choose one index (say j) that accesses them and eliminate any other index that accesses a location in public memory also accessed by j. This is repeated iteratively (for the analysis, we think of this as being done from the heavy public memory locations to the lighter ones). After the checker cannot be restricted any more we are left with a checker for which no two indices access the same location in public memory, and we will show that the number of remaining indices is reasonably high.

More concretely, for any value $i \in [1 \ldots \log\alpha]$, we know that there are at most $2^i - 1$ locations that are between $t/2^i$-heavy and $2t/2^i$-heavy. In fact, in the iterative restriction process, when we consider i we have already restricted the memory checker so that no location in the public memory is more than $2t/2^i$-heavy.

We go over these (at most $2^i - 1$) locations one by one, say in lexicographic order. For each of them, we examine one index that accesses that location, say index j. We restrict the checker by eliminating all "intersecting" indices: indices k such that there is a public memory location queried by both j and k. Index j queries at most q locations in the public memory, and these in turn are queried by at most $2t/2^i$ indices each (since we have already restricted the checker

so that there is no $2t/2^i$-heavy location in the public memory). Thus, we eliminate at most $2t \cdot q/2^i$ indices per heavy location in the public memory, or at most $2t \cdot q$ indices in all.

Repeating this for $i \leftarrow 1 \ldots \log \alpha$, in the i-th iteration there are at most 2^i locations that are at least $t/2^i$-heavy, and none of these locations can be more than $2t/2^i$-heavy. We go over these locations one by one, and if they have an index accessing them that has not been eliminated yet we restrict the checker as above. This eliminates at most $2t \cdot q/2^i$ indices per heavy public memory location, or $2t \cdot q$ indices in all.

In total, in all of these $\log \alpha$ iterations, with their restrictions, the number of indices eliminated is at most:

$$\sum_{i=1}^{\log \alpha} 2t \cdot q = 2t \cdot q \cdot \log \alpha$$

If $n > 4t \cdot q \cdot \log \alpha$ then we have only eliminated at most $n/2$ indices. Now, after all the restrictions, there are no locations in the public memory that are t/α-heavy. We go over the remaining indices in lexicographic order, and for each of them we restrict the checker by eliminating all other indices that intersect its public memory accesses. Since there are no more t/α-heavy locations in the public memory, each such restriction eliminates at most $q \cdot t/\alpha$ indices.

In the end, we are left with a memory checker on at least $n \cdot \alpha/(2q \cdot t)$ indices, with the property that no two indices access the same location in public memory. We can thus re-order the public memory into "chunks", of q symbols each, such that each chunk is queried only by a single index and each index queries only that chunk. If we enlarge the alphabet to be comprised of these q-symbol chunks, we get a checker with query complexity 1. The "price" is restricting the checker to only $n \cdot \alpha/(2q \cdot t)$ indices and increasing the alphabet size to Σ^q. Since we have increased the alphabet size, we can represent the secret memory as fewer symbols of the new larger alphabet, so the secret memory is of size s/q new alphabet symbols. □

of Lemma 3.3. The intuition is that the public memory has a single location for storing information about each database index. When reading or writing the value of the database at that index, the only information read from public memory is the information held in that index's location. Further, for two different database indices, their locations in public memory are different. To achieve soundness the checker must (intuitively) store, for every index in the database, separate "authentication information" in the secret memory about the value at that index's location. There are n indices (say holding boolean data base values), and only $s \cdot \log |\Sigma|$ bits of secret memory, and thus s should be at least on the order of $\frac{n}{\log |\Sigma|}$.

To prove this we examine an adversary $\mathcal{A}$, who begins by storing the all 0 database into the memory checker. This yields some public memory $\vec{p_1}$. $\mathcal{A}$ then picks a random database $\vec{r} \in \{0,1\}^n$ and stores it into the checker: for every index in $\vec{r}$ which has value 1, $\mathcal{A}$ uses the checker to store the value 1 into that index. Say now that at the end of this operation sequence, the public memory is $\vec{p_2}$ and the secret memory is $\vec{s_2}$. The important thing to note is that for indices of r whose values are 0, the value of their locations in the public memory has not changed between $\vec{p_1}$ and $\vec{p_2}$ (since each index has a unique location in public memory that it accesses).

The adversary $\mathcal{A}$ now replaces the public memory $\vec{p_2}$ with the "older" information $\vec{p_1}$.[3] Now the adversary tries to retrieve some index of the database, say the i-th ($i \in [n]$). The checker runs with secret memory $\vec{s_2}$ and public memory $\vec{p_1}$ to retrieve the i-th bit of $\vec{r}$. Note that if $\vec{r}[i] = 0$, then the value of the i-th index's location in public memory is *unchanged* between $\vec{p_1}$ and $\vec{p_2}$. By *completeness*, the checker should w.h.p. output 0 (the correct value of $\vec{r}[i]$). On the other hand, if $\vec{r}[i] = 1$, then by its *soundness* guarantee the memory checker should w.h.p. output either 1 or $\perp$ - we take either of these answers as an indication that $\vec{r}[i] = 1$. We conclude

[3]Note that this is a "replay attack". As noted above, the Lemma and this section's query complexity lower bounds do not hold for checkers that are not required to work against replay attacks.

that for each index $i \in [n]$, the checker can be used to retrieve the i-th bit of $\vec{r}$ w.h.p. The checker achieves this using only the public memory $\vec{p_1}$, which is completely independent of $\vec{r}$, and the secret memory $\vec{s_2}$. Intuitively, $\vec{s_2}$ holds nearly all the information about the (randomly chosen) vector $\vec{r}$, and thus $\vec{s_2}$ cannot be much smaller than $\vec{r}$, an n-bit vector.

More formally, suppose that $s < \frac{n}{\log|\Sigma|} - 1$. We can view the above procedure as allowing us to transmit a random n-bit string using only $s \log|\Sigma|$ bits and succeeding with high probability: the sender and the receiver share the initial assignment to the secret memory $\vec{s_1}$ and the public memory $\vec{p_1}$ resulting from writing the all 0 vector (all this is independent of r). Given the string $\vec{r} \in \{0,1\}^n$ the sender simulates writing $\vec{r}$ to the memory as above and the resulting secret memory at the end is $\vec{s_2}$. This is the only message it sends to the receiver. The receiver runs the above reconstructing procedure for each $1 \leq i \leq n$, i.e. using secret memory $\vec{s_2}$ and public memory $\vec{p_1}$ tries to read location i and decides that $\vec{r}[i] = 0$ iff it gets as an answer a 0 (1 or $\perp$ are interpreted that $\vec{r}[i] = 1$). Since for each i the procedure the receiver is running is just what the memory checker will run with the above adversary, the probability of error in any of the i's is small. Therefore we get that the receiver reconstructs all of $\vec{r}$ correctly with high probability. But by simple counting this should happen with probability at most $\frac{2^{s \log|\Sigma|}}{2^n} < 1/2$.

□

4 Read-Write Tradeoffs for Online Checking

In this section we present two read-write tradeoffs for the query complexity of online memory checking. These can be viewed as counterparts to the lower bound of Theorem 3.1 (all of the memory checkers in this section are deterministic and non-adaptive). While Theorem 3.1 states that the *sum* of the query complexities of read and write operations cannot be low, in this section we show that the query complexity of either read or write *can* be made significantly lower, at the cost of increasing the query complexity of the other operation (write or read respectively).

We present two trade-offs. The first gives an memory checker with efficient write operations but expensive read operations. The second is a checker with efficient read but expensive write. In particular, in both these tradeoffs, for any well-behaved function $d(n) : \mathbb{N} \to \mathbb{N}$, the "efficient" operation (write or read) has query complexity $O(\log_{d(n)} n)$, and the "inefficient" operation (read or write respectively) has query complexity $O(d(n) \cdot \log_{d(n)} n)$. In both cases the space complexity is polynomial in the security parameter, and the checker uses a pseudo-random function. For desired soundness ε the length of alphabet symbols is $O(\log(1/\varepsilon) + \log n)$.

Overview of the Constructions. We proceed with an overview of the common elements of both constructions, the details are below. Following Blum *et al.* (Section 5.1.2), we construct a tree structure "on top" of the memory. Where they constructed a binary tree, we construct instead a $d(n)$-ary tree. Each internal node has $d(n)$ children, so the depth of the tree is $\log_{d(n)} n$. The n leaves of the tree correspond to the n database indices. We assume for convenience w.l.o.g that n is a power of $d(n)$.

In both constructions we associate a time-stamp with each node in the tree. The time-stamp of a leaf is the number of times that the user wrote to the database index that the leaf represents. The time-stamp of an internal node is the sum of its children's time-stamps, and thus the time-stamp of the root is the total number of times that the user has written to the database. We use t_u to denote the current time-stamp of tree node u. The time-stamps are used to defeat *replay attacks* (where the adversary "replays" an old version of the public memory). If the adversary replays old information, then the replayed time-stamps will have smaller values than they should.

For each tree node u, we store in public memory its value $v_u \in V$ and its time-stamp $t_u \in [T]$. For an internal node u, its value is simply 0, for a leaf ℓ, its value represents the value that the user stored in the database index associated with that leaf. The root's time-stamp is stored in the secret reliable memory, together with the seed of a pseudo-random function (PRF). This simply a generalization of Blum *et al.*'s construction (the tree is $d(n)$-ary and not binary).

Our two construction differ from each other and from [BEG+94] in their use of authentication tags to authenticate different nodes' values and time-stamps. In the **first construction** (efficient write), we store for each node u an authentication tag which is the PRF evaluated on (u, t_u, v_u). When writing a new value to a leaf, we verify the tags of all the nodes on the path from the root to that leaf and then update the leaf's value and the time-stamps of all the nodes on the path to the leaf. Thus the write complexity is proportional to the tree depth, or $O(\log_{d(n)} n)$. To read the value from some leaf, we read the values, time-stamps and tags of that leaf, all nodes on the path from the root to the leaf and all their children, a total of $O(d(n) \cdot \log_{d(n)} n)$ public memory locations. We verify the consistency of all the tags, and that the time-stamp of every internal node is the sum of its children's time-stamps. This prevents replay attacks, as the root's time-stamp is in the reliable memory and thus always correct. The **second construction** (efficient read) is different. For each tree *edge* connecting a node u and one of its $d(n)$ children w, we store in public memory a tag which is the PRF evaluated on $(u, t_u, v_u, w, t_w, v_w)$. Now, to read the value from a leaf we read the values and time-stamps of all nodes on the path from the root, and the tags of the edges. For each edge we verify that the tag is consistent. This requires making $O(\log_{d(n)} n)$ queries to public memory. To write a new

value to a leaf, read and write the values and time-stamps at the leaf and all nodes on the path from the root to the leaf, as well as all their children and edge tags, a total of $O(d(n) \cdot \log_{d(n)} n)$ queries. Verify that all tags are consistent and that the time-stamp of each internal node is the sum of its children's time-stamps. If all checks pass, update the proper time-stamps and the leaf's value. We proceed with full descriptions of the constructions.

Notation and Assumptions. Fix n and take $d = d(n)$. Following the notation of [BEG+94], we take T to be an upper-bound on the number of system operations, V the set of values that can be written into a database index.[4] Throughout this section we take $\kappa(n)$ to be a security parameter and we assume the existence of a one-way function that cannot be inverted on inputs of length $\kappa(n)$ by poly(n)-time adversaries. In particular, let ε be the desired soundness, and take $\ell = \log(1/\varepsilon)$. We will use a family of pseudorandom functions $F = \{f_s\}_{s \in \{0,1\}^\kappa}$, where each function has range $\{0,1\}^\ell$. We choose κ such that no polynomial PRF adversary has noticeable advantage in distinguishing (from black-box access) a randomly chosen $f_s \sim F$ from a truly random function. In particular, no efficient adversary (with black-box access to f_s) can predict f_s's value on any previously unseen input with probability noticeably greater than $1/\varepsilon$. The size of the checker's alphabet will be $O(T + |V| + 1/\varepsilon)$.

4.1 Efficient Write

The Construction. In this section we design a checker with cheaper write complexity at the cost of more expensive read complexity. We will use a family of pseudorandom functions $F = \{f_s\}_{s \in \{0,1\}^\kappa}$, where each function f_s is from $[2n] \times [T] \times V$ to $\{0,1\}^\ell$. The checker chooses a random function from the PRF collection by selecting a random key s, and stores that key in the secret memory. For every node u with value v_u and time-stamp t_u, the checker stores an authentication tag $f_s(u, t_u, v_u)$ in public memory.

Reading. To read the value in index i of the database, the checker reads the value, time-stamp and tag of the leaf corresponding to that database index, as well as the values, time-stamps and tags of all the nodes on the path from the leaf to the root and all of their children. This requires reading a total of $O(d(n) \cdot \log_{d(n)} n)$ public memory locations. The checker then verifies that the tags are all consistent with their respective values and time-stamps, and that the time-stamp of every internal node is the sum of its children's time-stamps. If this is not the case then the memory checker rejects, otherwise it returns the value read from i's leaf.

Writing. To write a value v to location i, the checker reads and writes the value, time-stamp and tag of location i's leaf, as well as the values, time-stamps and tags of all the internal nodes on the path from that leaf to the root (without reading or modifying any of their children). The checker first verifies that all the tags it read were valid (otherwise it rejects), and then updates the leaf's value, increases by 1 the time-stamps of the leaf and all the nodes on the path from the leaf to the root, and updates all of these nodes' tags with the new time-stamps (and the leaf's with the new value). This requires reading and writing to a total of $O(\log_{d(n)} n)$ locations in the public memory.

Security Analysis. Because the checker uses a pseudo-random function, which in particular is also unpredictable, no adversary can modify any individual node's value and time-stamp to any combination that has not occurred before, while also avoiding detection by generating a legal tag (with probability noticeably greater than ε, the probability of "guessing" the relevant tag).

[4] In this work we focus on binary memory checkers, where $V = \{0,1\}$, but this is easily generalized.

This leaves the question of replay attacks, in which the adversary cheats by *replaying* an old value time-stamp combination for a node. In particular, the replayed time-stamp must be strictly smaller than the correct one (as the time stamp is increased every time there is a write to a node's descendant). Say that we read the value of index i and the adversary replays an old value and time-stamp. The root's time-stamp (and value) are in secret memory and we are thus guaranteed that they are correct. There must be some "lowest" node on the path from the root to i's leaf whose time-stamp is correct, but one of whose children's time-stamps is too small. The checker verifies that each node's time-stamp is the sum of it's children's, so the time-stamp read for one of that node's children must be higher than it should be. The checker verifies all children's tags, and so this means that to fool the checker the adversary has to predict the tag on a previously unseen time-stamp! Such an adversary could break the pseudo-random function.

We note that during *write* operations the adversary *can*, in fact, replay the values and time-stamps of tree nodes without being caught (since we do not check that each node's time-stamp is the sum of its children's). This is not a problem: such modifications do not compromise the checker's security. This only allows the adversary to obtain tags for time-stamps that are smaller than they should be, but it *will still be caught* if it tries to give incorrect values or time-stamps during read operations. This insight allows us to avoid checking the tags of all the children of each internal node during write operations, and thus to obtain a reduced write complexity.

4.2 Efficient Read

The Construction. In this section we construct an online memory checker with cheaper read complexity at the cost of more expensive write complexity. In addition to storing the tree leaves, the checker uses the public memory to store information about the tree's internal nodes and *edges* (Blum *et al.* and the construction of Section 4.1 only store information about the tree nodes). For each tree *edge* j connecting a node u and one of its d children w, we store in public memory an authentication tag $f_s(u, t_u, v_u, w, t_w, v_w)$.

Reading. To retrieve the value at the database's i-th index, the checker reads the value stored at the leaf corresponding to that index as well as all the tags and values on the path (nodes and edges) from that leaf to the root. This requires reading a total of $O(log_d n)$ locations in public memory. The checker verifies that for each pair of nodes (u, w) along the path, if the values and time-stamps read for these nodes were (respectively) v'_u, t'_u and v'_w, t'_w, then the tag stored for the edge (u, v) is indeed, as it should be, $f_s(u, t'_u, v'_u, w, t'_w, v'_w)$.

Writing. To modify the value stored at the database's i-th index (a *write* operation), the checker reads and writes the value and time-stamp stored at the leaf corresponding to that index, as well as all the values and time-stamps on all the internal nodes on the path from the root to that leaf. In addition, for each such internal node, the checker reads the tags of *all the edges* leading to its children and the time-stamps and values of *all its children*. This requires accessing (reading and writing) a total of $O(d \cdot log_d n)$ locations in public memory. The checker first verifies that all the tags read are consistent with the time-stamps and values of their respective nodes, and that the time-stamp of each of the internal nodes is the sum of its children's time-stamps (the memory checker rejects if this is not the case). The time-stamps of the nodes on the path from the leaf to the root, and the leaf's value, are then updated (each time-stamp is increased by 1), and finally the checker updates all of the authentication tags accordingly.

Security Analysis. As above, the checker uses a pseudo-random function, which is in particular also unpredictable. This means that no adversary can modify any pair of adjacent nodes' values and/or time-stamps to any combination that it has not seen before for that particular

pair, while also avoiding detection by generating a legal tag (with probability noticeably greater than ε, the probability of successfully "guessing" the relevant tag).

This leaves the issue of replay attacks. Suppose an adversary successfully replays old values or time-stamps during a read or write operation. The time-stamp of the root is in the (reliable) secret memory, so there must be some lowest node that is accessed during the operation whose time-stamp is read correctly, but one of whose children has a lower time-stamp than it should. The combination of the node's time-stamp and that of its child has not occurred before (because the node's time-stamp is increased whenever one of its children's time-stamps is increased). Thus, either the tag of the edge from the node to its child is illegal, and the checker will reject, or the adversary has successfully predicted the value of the pseudo-random function on a point it has not seen before. If this happens with probability noticeably greater than ε, then this adversary can be used to break the pseudo-random function.

5 Offline Checking of RAMs

In this section we describe how to check "offline" the operation of a RAM, that is a sequence of read and write (or store and retrieve) operations. To check that a RAM operates correctly we must verify that the value we obtain from reading an address in public memory is equal to the last value we wrote to that address. Blum *et al.* [BEG+94] showed an (invasive) scheme, where if one scans the whole memory at the end of the sequence of operations, then it is possible to detect (with hight probability) any malfunction. The cost (in query complexity) is $O(1)$ per operation, plus the final scan. Thus, for sequences of n operations or more, the *amortized* query complexity is $O(1)$. As discussed in the introduction, our goal is to improve upon that, by not running a final scan of all the memory. Instead, we scan only the locations that were changed. This implies that at any point, after t operations, we can check that the memory worked appropriately by investing time $O(t)$, so for *any* sequence of operations (not only for long ones) the amortized query complexity is $O(1)$. This result can be viewed as a generalization of those in Amato and Loui [AL94].

Our ideas follow closely those of Blum *et al.* [BEG+94]. First, add to each memory address a slot for the time it was written - a "timestamp". The "time" can be any discrete variable that is incremented whenever a write or read operation is performed. The timestamp of each location is actually updated after either read or write. So one can view each operation as read followed by write. The offline checker needs to verify that the set of (value, address, time) triples which are written equals the set of (value, address, time) triples which are read. More precisely, consider the following two sets:

$$R = \{(v, a, t) | \text{location } a \text{ was read with value } v \text{ and timestamp } t\}$$

$$W = \{(v, a, t) | \text{location } a \text{ was written with value } v \text{ and timestamp } t\}$$

Suppose that at no point in time did a read operation return a timestamp larger than the current time (call this the timestamp condition), a clear malfunction, and suppose that the memory is scanned (i.e. read completely) at the end of the sequence of operations. Then Blum *et al* [BEG+94] showed

Claim 5.1. *$W = R$ iff the memory functioned properly.*

In other words, a procedure that checks online for the timestamp condition plus an offline test for $W = R$ results in an offline checker for the RAM. It is useful to note that the proof actually implies that if the timestamp condition was not violated, then actually $W \not\subseteq R$.

We modify slightly the above and note that if we scan *only those locations that were actually modified*, then we can similarly say that $W = R$ iff the memory functioned properly. This is true, since the locations that were not written do not affect W and hence whether we access them or not does not make $R = W$.

Now the question is, how do we scan only the locations that were modified? For this we keep a linked list of all locations that were accessed. When a new location is read it is added to the end of the list. The starting and ending locations of the list are stored in the secure memory. To scan the locations accessed we trace the list, see below on possible implementations of the list, the important thing is that adding a memory location to the list and checking whether a memory location is already in the list can be done with $O(1)$ queries (possibly amortized).

A natural question now is how to authenticate the list to ensure that an adversary did not tamper with it (i.e. *who guards the guard?*). The point here is that *the list itself need not be authenticated*. To address the issue of faults in the linked list, observe that as indicated above to make the checker accept the adversary needs to "cover" W by R. If the adversary tampers with the list, and a wrong set of locations is accessed in the final scan, then it will not cover W. Since we do not authenticate the list, the one remaining concern is that a faulty memory can

even lead the scanning process into loops (by putting loops into the list). To prevent this, we use a simple counter that bound the number of locations we traverse in the list.

To check whether W and R are the same or not, we can use the same methods as described in Blum et al. The problem can be thought of as one in *streaming*, since the sets are accessed one value at a time. We assume there is a secret hash function h mapping sets to some range and we can compute on the fly $h(R)$ and $h(W)$ and compare the results. That is, h can be updated incrementally in each operation (read or write).

Specifically, we require that where for every k there exists a family H where: (i) representing a member $h \in H$ of the family takes $O(k + logn)$ bits (ii) the range of $h \in H$ can be represented by $O(k)$ bits (iii) the probability that two different sets hash to the same value is at most $1/2^k$ (the probability is over the choice of the function in the family) and (iv) given $h \in H$ an element x and the value of $H(S)$, the value of $h(S \cup \{x\})$ can be computed in $O(1)$ operations. There are constructions of such functions (see e.g. Naor and Naor [NN93] and the analysis in [BEG$^+$94]). The procedures for reading and writing are now as follows:

Write of value v to address a

- read the value v' and time t' stored in address a.
- verify that t' is less than the current time.
- update the hash $h(R)$ of set R with (v', a, t').
- write the new value v and current time t to address a.
- update the hash $h(W)$ of set W with (v, a, t).
- if location a is not in the linked list add it to the end and update the endpoint in the secure memory.

Read of address a

- read the value v' and time t' from address a.
- verify that t' is less than the current time t.
- update the hash $h(R)$ of set R with (v', a, t').
- write v' and t to address a.
- update the hash $h(W)$ of set W with (v', a, t).
- if location a is not in the linked list add it to the end and update the endpoint in the secure memory.

To check the functioning of the RAM at the end of any sequence of operations, the checker reads all the memory locations in the linked list, starting from the first location in the list, which is stored in the secure memory. As the scan proceeds $h(R)$ is updated accordingly. Assuming initially $W = R = 0$ and the RAM is empty, $h(W)$ should equal $h(R)$ if the memory functioned correctly, and should be different from $h(R)$ with high probability if the memory was faulty. To maintain the list of modified locations, we can use a simple linked list (see below for a more efficient alternative). It is enough to add a pointer to each address in public memory (together with the value and timestamp of that address). The pointer is initially NULL (or 0), and whenever we access a public memory location for the first time we modify the pointer of the current list tail to point to the new list end and update the list end (there is no need to update R and W for list maintenance operations, faults in the list will be detected).

Note that we do not have to assume that the memory is initialized to be all 0 before the beginning of the operations, since it is possible to use the "uninitialized memory trick", where one keeps a list of pointers to the modified locations and all other locations are 0. See [AHU74], exercise 2.12 or [Ben86, Cox, BT93].

Since the scheme is invasive (has to change the memory), it makes the most sense when the basic unit we read is relatively large. Suppose the length of a database word is μ, then the

additional timestamp takes $\log n$ bits and the pointer to the linked list takes another $\log n$ bits. We summarize the results in the following theorem.

Theorem 5.1. *For a RAM with n words of size μ there exists an invasive, offline memory checker using n memory locations storing $\mu + 2\log n$-bit words, which uses $O(\log n + \log 1/\varepsilon)$ private memory. Each read or write operation takes $O(1)$ queries, and a procedure for detecting error can be executed after any sequence of t steps at the cost of $O(m)$ where m is the actual number of locations that were used. An error is detected with probability at least $1 - \varepsilon$.*

Finally, we re-examine the issue of invasiveness. We note that in fact we do not need to store time-stamps and list-pointers for *all* of the database indices, just for those that are accessed. This leads to a method for reducing the invasiveness of the checker (the total number of non-database bits that it stores in public memory). We can maintain the timestamps and the list itself as a separate data structure, whose size is proportional (say linear) to the number of database indices which have been accessed. Any data structure that supports insertion and membership queries in amortized $O(1)$ time work. We note once more that Ajtai [Ajt02] proved a lower bound on the invasiveness of offline memory checkers, but his proof uses long sequences of operations that access every database index, and thus it does not apply to our setting of short sequences of operations that access only a few locations in the database.

References

[ABC+07] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. Cryptology ePrint Archive, Report 2007/202, 2007.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms (Addison-Wesley Series in Computer Science and Information Processing)*. Addison Wesley, January 1974.

[Ajt02] Miklós Ajtai. The invasiveness of off-line memory checking. In *STOC*, pages 504–513, 2002.

[AL94] Nancy M. Amato and Michael C. Loui. Checking linked data structures. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 164–173, 1994.

[BEG+94] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[Ben86] Jon Bentley. *Programming Pearls*. ACM, New York, NY, USA, 1986.

[BT93] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2:59–69, 1993.

[Cox] Russ Cox. http://research.swtch.com/2008/03/using-uninitialized-memory-for-fun-and.html.

[CSG+05] Dwaine E. Clarke, G. Edward Suh, Blaise Gassend, Ajay Sudan, Marten van Dijk, and Srinivas Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE Symposium on Security and Privacy*, pages 139–153, 2005.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali. How to construct pseudorandom functions. *Journal of the ACM*, 33(2):792–807, 1986.

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1*. Cambridge University Press, 2001.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2*. Cambridge University Press, 2004.

[JK07] Ari Juels and Burton Kaliski. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597, New York, NY, USA, 2007. ACM.

[NN93] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comput.*, 22(4):838–856, 1993.

[NR05] Moni Naor and Guy N. Rothblum. The complexity of online memory checking. In *FOCS*, pages 573–584, 2005.

[OR07] Alina Oprea and Michael K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *USENIX Security Symposium*, 2007.

[SW08] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *ASIACRYPT*, pages 90–107, 2008.

A Ajtai's Invasiveness Lower Bound

In this Section we review Ajtai's invasiveness lower bound for offline memory checkers. Ajtai showed that there are no non-invasive offline checkers with logarithmic space complexity (even if one allows small polynomial query complexity). Beyond this, he even showed a lower bound on the amount of extra invasive information that the checker needs to store in public memory. Ajtai defined an α-invasive checker as a checker that adds α bits of invasive information per database index. It is assumed that this additional invasive data is accessed only when reading or writing to its index (as in the case both in the construction of [BEG+94] and in our construction above). Formally, he showed:

Theorem A.1 (Ajtai [Ajt02]). *For every $c > 0$ there exists $\varepsilon > 0$ such that there is no $\varepsilon \cdot \log n$-invasive offline memory checker with space complexity at most $c \cdot \log n$ and soundness $1/n$. This lower bound holds even if one considers only* efficient *adversaries.*

In this work we show that the amount of invasiveness can be reduced if one considers short operation sequences: the invasiveness can be proportional to the number of database indices accessed by the user. This does not contradict Ajtai's theorem, since his theorem (and proof) only lower-bound the amount of invasiveness required for *long* sequences of operations that access all the database indices.